
# DOGE HOUSE Token Audit

Conducted by Meme Audit

Contract Address: 0x61B49c26334397c16624453F0d98547F801329C2

Chain: Polygon (MATIC)

# Smart Contract Overview

The DOGE HOUSE token contract is a standard ERC-20 implementation built on the Polygon chain. It follows OpenZeppelin v5.4.0 standards, ensuring high code quality, security, and maintainability.

Key Token Information:

- Token Name: DOGE HOUSE
- Symbol: DOGE HOUSE
- Total Supply: 1,000,000,000 (1B)
- Decimals: 18
- Standard: ERC-20
- License: MIT

## Security Highlights

- No public mint function shown in the excerpt.
- Uses OpenZeppelin ERC20 baseline and modern Solidity ^0.8.20 practices.
- No external low-level calls or reentrancy-sensitive logic shown.
- Uses custom errors (ERC-6093) for gas-efficient error reporting.

# Audit Results

Summary:

- Security Score: 95 / 100 (provisional based on provided code excerpt)
- Risk Level: LOW (assuming no hidden owner mint/backdoor functions exist in omitted sections)

Notes:

- This score is provisional because the provided code is truncated. The final score depends on constructor and any additional functions not included in the excerpt.

# Detailed Solidity Findings

Summary of Analysis (based on provided Solidity code excerpt, ERC20 OpenZeppelin v5.4.0):

- Contract Structure:
The code follows OpenZeppelin's ERC20 implementation (Context, IERC20, IERC20Metadata, ERC20). That's a strong baseline for correctness, safety, and compatibility.

- Minting & Supply:
The ERC20 implementation's _update allows minting when `from == address(0)` (i.e., _mint uses _update). The excerpt does not include the contract constructor or any mint/burn functions. If the contract mints total supply only in the constructor and no external mint function exists, minting risk is low. Confirm the constructor mints exactly 1,000,000,000 * 10**18 to the intended owner and that there are no other mint entry points.

- Ownership & Admin Privileges:
The excerpt does not show Ownable or similar owner-control logic. If the full contract includes Ownable or an admin role, review those functions carefully (transferOwnership, renounceOwnership, owner-only functions). If not included, the token is a pure ERC20 and less likely to have centralized backdoors but confirm there are no hidden external calls later in the code.

- Approve / Allowance Behavior:
This OpenZeppelin implementation uses the standard approve/transferFrom pattern and includes the standard note about allowance race conditions. This is expected, but you may wish to use safe patterns (increaseAllowance/decreaseAllowance) in user-facing docs.

- External Calls & Reentrancy:
There are no external payable or low-level calls in the provided code. Pure ERC20 transfer/transferFrom logic is not reentrancy-prone. If the full contract adds functions that call external contracts, those must be reviewed for reentrancy and checks-effects-interactions pattern.

- Integer Safety:
Solidity ^0.8.x includes built-in overflow checks; OpenZeppelin uses unchecked blocks where safe. This is fine.

- Gas / Efficiency:
The code uses mappings and conventional patterns. Consider whether metadata() or other view functions are required. No major gas anti-patterns were seen in the excerpt.

- ERC-20 Standard Compatibility:
The functions and events (Transfer, Approval, name, symbol, decimals, totalSupply, balanceOf, transfer, approve, transferFrom, allowance) are present and implemented in a standards-compatible way.

- Error handling:

The code uses custom errors per ERC-6093 (e.g., ERC20InsufficientBalance) which is gas-efficient and modern. That's a good practice.

Recommendations (actionable items):
1. Confirm constructor and supply minting: verify that total supply (1,000,000,000 * 10**18) is minted once in the constructor to the correct deployer/treasury address, and no public/external mint function exists.
2. If ownership/admin exists: ensure owner renounce/transfer functions are implemented correctly and consider renouncing ownership or using multisig timelock for admin actions.
3. Add Liquidity Lock: if liquidity will be added on a DEX, lock liquidity (or document lock) and provide proof of lock (e.g., UniSwap/Quickswap lock) to increase trust.
4. Add token recovery or emergency functions only if necessary and make them time-locked and multisig protected.
5. Publish unit tests and formal verification outputs: include tests for transfer, approve, transferFrom, allowances, and edge cases (zero address transfers, allowance exhaustion).
6. Provide deployment & constructor parameters: include exact constructor mint amount and recipient address in the report for transparency.
7. Add thorough README & metadata on Polygonscan: ensure constructor arguments and source match verified bytecode.
8. Recommend adding a burn function only if intended and, if present, document its behavior and limits.
9. Consider implementing increaseAllowance/decreaseAllowance helpers in front-facing UI to avoid the approve race-condition issues.
10. Consider adding ERC-20 permit (EIP-2612) only if gasless approvals are required by your ecosystem.

Risk Summary:
- Based on the excerpt alone, no high-severity vulnerabilities were detected.
- The primary risks to check in the missing parts of the contract are any owner-only mint/burn functions, external call wrappers, or admin functions that can manipulate balances or allowances.

Conclusion:
- Assuming the full contract matches the OpenZeppelin ERC20 baseline and the only minting happens once at deployment, the contract is low-risk for standard ERC-20 operations. Confirm the missing parts (constructor and any additional functions) to finalize the assessment.

**Certified by Meme Audit**